

A Model of bCMS Using the Umple Model-Oriented Programming Approach

Timothy Lethbridge

Electrical Engineering and Computer Science

University of Ottawa, K1N 6N5 Canada

+1 613 562-5800 x6685

tcl@eecs.uottawa.ca

ABSTRACT

In this paper we present a model developed using the Umple model-oriented programming technology. We have modeled the bCMS crisis management system, which has been set as a common modeling challenge for the Comparing Modeling Approaches workshop. We demonstrate modeling of the class structure of the system and the state machine used to co-ordinate the crisis management effort. This model captures some of the benefits of model-oriented programming: The model looks like a program, so it can be extensively documented inline using comments, and easily edited. The textual model is designed to be highly readable, but if desired can also be viewed and edited as a diagram. Mixin and product-line capabilities of Umple are also used.

General Terms

Design, Language

Keywords

Umple, Model-Oriented Programming, UML.

1. INTRODUCTION

We present a model of the bCMS system [1] developed using the Umple model-oriented programming technology.

We start with an overview of what we mean by model-oriented programming and a few facts about Umple and the modeling approach in this paper. The remainder of the paper shows the model.

1.1 Model-Oriented Programming

Model-oriented programming is a modeling approach that has the following properties:

- Model and code are a unified textual artifact that can be version controlled, edited in a text editor, and operated on by any standard tool that works with text. The textual notation is designed to be as readable by humans as possible, as such XML, the basis for many textual modeling languages, has been avoided.
- Modeling constructs are added as first-class entities and as parsimoniously as possible to existing programming languages, such as Java, PHP, C++ etc.
- All modeling constructs result in high-quality generated code when compiled, in the same manner as the existing language constructs generate bytecode or machine code. So-called ‘round trip engineering’ (editing of generated code with propagation of changes to the source) is conceptually

unnecessary and strongly discouraged, since any needed editing can be done in the original source.

- The processes of modeling and programming are also unified: Programming is seen simply as development at a lower level of abstraction, while modeling is seen as development at a higher level of abstraction. We will refer to the people who do the modeling/programming task as developers, to be generic
- A program developed using model-oriented programming at any level of abstraction can be ‘partial’, resulting in a sketch, or incomplete system. However, as long as it is syntactically and semantically correct, even a partial model-oriented program can be compiled; an incomplete model would generate an API, for example.
- The types of modeling constructs are not limited by the approach. Currently implemented examples include class diagrams, state diagrams, constructs for databases (keys), and various design patterns.
- A visualization of the model/program may be available. In such a case, the text and diagrams are views of the same underlying model/program.
- A developer can create a ‘pure’ model that has only modeling constructs. When doing this he or she is modeling textually. On the other hand a developer might start with a program in a standard programming language, and convert it to a model, by transforming parts of the code into modeling elements with a higher level of abstraction. Doing so would typically reduce the volume of code substantially (by between 10% and 90% depending on the case [2]).

1.2 Umple: An Example of a Model-Oriented Programming Technology

We have been developing Umple [3][4][5] since 2007 as a technology to explore the ideas of model-oriented programming. Umple has support for UML class diagrams and state machines, plus aspect-orientation and certain design patterns. Various other features are under development, particularly extensions for UML component diagrams and real-time profiles.

Umple supports product lines and feature-driven development in two ways: It has a mix-in capability that allows constructing a system from different configurations of parts. This capability applies to the base language (e.g. Java) too, so for example different Java methods can be mixed in to give a different version of the same class for two different members of a product line. Umple also has an implementation of VML (Variability Modeling Language) [6] that can be used to put together different configurations of features; it leverages the mixin capability.

Umple has been used to create a wide variety of systems, including the Umple compiler itself. There is an online tool [7]

```
// RoutesAndLocations.ump
//
// Author: Timothy C. Lethbridge

namespace routesAndLocations;

/*
 * This file describes reusable model elements
 * that are used in the bCMS Crisis Management
 * System. It is necessary to have a rudimentary
```

```

* Graphical Information System such as described
* in this submodel as part of bCMS in order to
* plan routes and to track where crises are.
* This file has the necessary classes.
*/

/*
* A CityMap contains the precompiled streets and
* landmarks
* This must be read in from the database on
* system initialization
*/
class CityMap {
    singleton;
    1 -> * Landmark fireStn;
    1 -> * Landmark policeStn;
    1 -> * Landmark otherLm;
    1 -> * NamedRoad;
}

/*
* A Location describes a place such as an
* intersection, landmark, bend in a road, etc.
*/
class Location {
    Float lattitude;
    Float longitude;
}

class Landmark {
    isa Location;

    // Name of landmark (name of fire station,
    // business, address)
    description;
    landmarkType { fireStation { } policeStation { }
    touristDestination { } other { } }
}

/*
* RoadNodes are places where RoadSegments
* connect. A node with just one incoming and one
* outgoing is used to handle changes in
* direction, e.g. as a road turns a corner,
* changes in speed limit, changes in number of
* lanes, and other factors.
*
* When there is more than one outgoing or
* incoming, the node represents points
* where traffic flow can split and join.
* A crossroads is one type. Entry into a
* roundabout would be another,
* Highway merges and exits are other kinds as
* are parking lots and entries into or out
* of fire stations..
*
* RoadSegments leaving the city lead to no
* RoadNode
* RoadSegments entering the city come from no
* RoadNode
*/
class RoadNode {
    Integer id;
    // Could be at a landmark
    0..1 -> 1 Location nodeLocation;
    0..1 end -- * RoadSegment incomingRoads;
    0..1 start -- * RoadSegment outgoingRoads;
}

class Intersection {
    isa RoadNode;
    // illegal, but possible turns for police
    // and fire

```

```

    1 -> * Turn illegalTurns;
    // Impossible turns , e.g. because of barriers,
    // turning radius
    1 -> * Turn impossibleTurns;
}

/*
* Turns are used to model illegal turns, e.g.
* turning left when there is 'no left turn'
* allowed, or 'no U turn' allowed.
*/
class Turn {
    RoadSegment turnFrom;
    RoadSegment turnTo;
}

/*
* A named road might have the name of a street,
* the number of a highway, etc. A RoadSegment
* can have several names because for example,
* sometimes several numbered highways share a
* segment
*/
class NamedRoad {
    name;

    // Most roads have two directions, e.g. North
    // and South
    direction;

    // Listed in order
    * -- 1..* RoadSegment segments;
}

/*
* A RoadSegment represents a section of road on
* which a vehicle can drive.
* The ends of each segment have been modeled
* using RoadNodes.
* Note that distance can be calculated from the
* speed limit, and locations
*/
class RoadSegment {
    Integer speedLimit;

    // Indicator of
    //real-time congestion; 0=blocked
    Integer currentReportedSpeed;
    Integer lanes;

    // The following are used to determine addresses

    Integer streetNumberAtStart;
    Integer streetNumberAtEnd;
    * -> * RoadSegment inverseSegments;
}

/*
* A route is a plan to get from one location to
* another
* It is built by algorithms that traverse the
* nodes and arcs of the map taking into account
* speed, congestion, etc.
*/
class Route {
    // Ordered list of segments
    1 -> 1..* RoadSegment;

    // Time seconds at current flow speeds
    Integer estimatedTime;
}

```

One of the interesting things to note about the Umlle model code is the ease of incorporating documentation of each model element

as code comments. These comments are highly visible, unlike in graphical presentations where tools usually require the user to hover over the element, or click on it. The comments are also preserved in the generated code, and are output in JavaDoc documents that might be used by a programmer to understand the generated API of the system. The reader is urged to actually generate the system and view the output.

Umple has a diagramming sublanguage that is used to describe the layout of diagrams. To save space, the diagram model is omitted in this paper, but it can be obtained via the UmpleOnline link mentioned above.

2.2 Submodel for Crisis States

The following is a model for the component representing the state machine used to co-ordinate the crisis. This can be accessed at: <http://try.umple.org?example=CoordinationStateMachine>, from which, once again, an executable of this subsystem can be generated.

A UML diagram for this state machine is shown in Figure 2. The state machine was generated using Papyrus, since UmpleOnline currently only supports class diagrams. Figure 2 omits some self-transitions to avoid excessive clutter.

```
// CoordinationStateMachine.ump
//
// Author: Timothy C. Lethbridge

/*
 * State machine for co-ordination of the
 * crisis, as seen from one SC. If
 * communication is operating properly the
 * other coordinators' state machines will
 * generally be in the same states and
 * substates.
 *
 * Modelled given Section 4 of the requirements
 * This is a mixin to class crisis.
 */
class Crisis {

    // Number of ms allowed to negotiate routes,
    // establish credentials
    Integer timeoutInMs = 20000;

    // True if we are the initiator of the crisis
    // request, false otherwise
    Boolean thisSideInitiator;
    Boolean thisSideInitiatorOfTermination;

    Boolean routeAgreedYet;

    // True of we are the PSC, false of we are
    // the FSC
    Boolean police;

    // Top level state machine
    crisisCoordinationStage {

        // 0. When initiated the crisis is in
        // noCrisis state
        // It has been created but not yet
        // populated, awaiting initiation
        // of coordination and data..
        noCrisis {
            entry / {setThisSideInitiator(false);}
            // We are initiating and requesting the
            // other side to respond
            initiateCrisis / {
                setThisSideInitiator(true);
            }
        }
    }
}
```

```
sendInitiateCrisisRequestToOtherSC();
} -> establishingCommunication;

// We are responding to a request from
// the other SC
receiveInitiateCrisisRequest ->
    establishingCommunication;
}

// 1. Establish communication and
// identification when a crisis begins
establishingCommunication {
    entry / {sendSecureCredentials();}
    secureCredentialsConfirmed ->
        managingCrisis;

    // Return to idle after some delay if
    // other side initiated and did not
    // respond. It might have been a
    // hacker. Alternatively if there is a
    // real crisis, normal manual processes
    // would happen.
    after (timeoutInMs)
        [!isThisSideInitiator()] -> noCrisis;
}

// Managing crisis involves continually
// exchanging information
// Negotiating or renegotiating routes and
// other crisis details
// These are handled concurrently.
managingCrisis {
    // A crisis is underway. We initially
    // start with empty data
    // which will be populated over time
    entry / {initiateEmptyCrisis(); }

    // 2,4,5,6. Exchanging details about what
    // each knows about the crisis
    // So the information each coordinator
    // has is the same.
    exchangingDetails {
        // Requirement Scenario 2
        ourUpdateToCrisisData
            / {sendCrisisData();}
            -> exchangingDetails;
        receiveCrisisData
            / {updateCrisisData();}
            -> exchangingDetails;

        // Requirement Scenario 4
        ourVehicleDispatched
            / {sendVehicleDispatch();}
            -> exchangingDetails;
        receiveVehicleDispatched
            / {updateTheirDispatch();}
            -> exchangingDetails;

        // Requirement Scenario 5
        ourVehicleArrived
            / {sendVehicleArrived();}
            -> exchangingDetails;
        receiveVehicleArrived
            / {updateTheirArrival();}
            -> exchangingDetails;

        // Requirement Scenario 6
        ourVehicleMetObjective
            / {sendVehicleMetObjective();}
            -> exchangingDetails;
        receiveMetObjective
            / {updateTheirMetObjective();}
            -> exchangingDetails;
    }
}
```

```

// Requirement 5.a - breakdown
breakdown / {dispatchAndUpdateOther();}
-> exchangingDetails;

// Requirement 5.b -
// congestion/blockage
// Omitted from this version

// Requirement 5.c escalate crisis -
// renegotiate routes
// Omitted in this version

} // End of exchangingDetails
// concurrent substate
||

// 7. Both parties must agree to close
// the crisis; either can initiate
crisisEndManagement {

    // Normal substate of crisisEndManagement
    // - no end in sight yet
    ongoingCrisis {

        // We could initiate termination
        initiateTermination / {
            setThisSideInitiatorOfTermination(
                true);
            sendTerminationRequestToOtherSC();
        } ->
            waitingForTerminationConfirmation;
        receiveTerminationRequestFromOtherSC
        ->
            waitingForUserTerminationAgreement;
    }

    waitingForUserTerminationAgreement {
        do {confirmWithUserToTerminate();}
        confirmTermination -> tearDown;
        after (timeoutInMs) -> ongoingCrisis;
    }

    // Substate of where we are waiting for
    // the other end to agree
    waitingForTerminationConfirmation {
        receiveTerminationConfirmation
        -> tearDown;

        // If the other side has not agreed,
        // we keep the crisis active
        after (timeoutInMs) -> ongoingCrisis;
    }

    // End of crisis
    tearDown {
        entry / {deleteCrisis();}
        -> noCrisis;
    }
} // End of crisisEndManagement
// concurrent substate
||

// 3. Negotiating route plan
negotiatingRoutePlan {

    // Negotiation happens in parallel with
    // reporting timeout
    negotiation {
        entry / {setRouteAgreedYet(false);}
        informNumberOfVehicles
        / {sendNumberOfVehicles();}

        -> negotiation;
        receiveNumberOfVehicles [isPolice()]
        -> planningRoute;
        receiveRouteProposal [!isPolice()]
        -> approvingRoute;

        // Requirement 3.3.a2.a1
        receiveNoAgreeableRoute
        [!isPolice()] -> noRouteAgreement;

        // The PSC plans the route -- only
        // PSC can be in this state
        planningRoute {
            do {planRoute();}
            routePlanComplete
            / {sendPlanToFSC(); }
            -> awaitingRouteApproval;

            // Requirement 3.3.a2.a1 - no more
            // possible routes
            noMoreRoutes
            / {sendNoMoreRoutesToFSC(); }
            -> noRouteAgreement;
        }

        // The FSC approves the route --
        // only FSC can be in this state
        approvingRoute {
            do {userConfirmRouteAcceptable();}
            routeAcceptable
            / {sendApprovalToPSC();}
            -> routeAgreed;
            routeUnacceptable
            / {sendDisapprovalToFSC();}
            -> negotiation;
        }

        // The PSC awaits for approval from
        // the FSC. Only the PSC can be in
        // this state
        awaitingRouteApproval {
            receiveApprovalFromFSC
            -> routeAgreed;

            // Requirement 3.3.a - FSC
            // disagrees
            receiveDisapprovalFromFSC / {
                addRouteToDisapprovedChoices();
            } -> planningRoute;
        }

        routeAgreed {
            entry / {setRouteAgreedYet(true);}
        }

        noRouteAgreement {
            // requirement 3.3.a2.a1
        }
    } // End of Negotiation concurrent
    // substate of negotiatingRoutePlan
    ||
    managingTimeliness {
        timeAcceptable {
            // Requirement 3.a1. Negotiations
            // are taking excessive time
            after (timeoutInMs)
            [!isRouteAgreedYet()]
            -> timeUnacceptable;
        }
        timeUnacceptable {
            // Although negotiations may yet
            // complete, we are going to send
            // out our own vehicles because too

```

```

        // much time has passed
    do {
        promptAndLogReasonForTimeout(); }
    }
} // End of managingTimeiness
//concurrent substate of
// negotiatingRoutePlan
} // End of NegotiatingRoutePlan
//concurrent substate
} // End of ManagingCrisis top level state
} // End of crisisCoordinationStage state
// machine
} // End of state machine mixin for
//StationCoordinator

```

The state machine can be compiled and will generate executable code. However the numerous methods (events and actions) would still have to be implemented. Event methods would be called either by the user interface or incoming communications from the ‘other SC’.

Figure 2 was generated for the above state machine using the Papyrus [12] open-source tool, since UmpleOnline is currently only capable of displaying class diagrams. The Umple textual form and the Papyrus visual form are essentially equivalent

Here are a few observations about the exercise of modeling this state machine in Umple, and then drawing it in Papyrus.

- The model has three top-level states, of which one (managingCrisis) has three concurrent regions. Two of these regions have multiple substates, and one has two additional levels of nesting. The Papyrus diagram makes this clearly visible, especially given that we have applied colouring. The nesting using braces) is harder to see in the Umple code.
- The Umple code, on the other hand, was far easier to edit and manipulate to get the model right. The Umple code also makes the embedded code for actions and the comments visible. Furthermore, we had considerable difficulty creating the diagram such that the transition labels were positioned in reasonable places. In one place, we replaced multiple transitions by the phrase ‘multiple transitions’ to avoid clutter

Overall this exercise helped demonstrate the value of having both visual and textual models that are concurrently visible and editable, as is the case with the class diagrams in UmpleOnline. Our experiences are also consistent with our recent experimental results [13] showing that users find visual and textual state machines equally useful and understandable, but perhaps for different reasons.

2.3 Submodel for the Central System Core

The central system core file contains a singleton class CrisisManagementSystem that is used to hold instances of the other classes.

It also contains the Crisis class. We have seen the state machine for this class in the previous section. Here we are mixing in the core attributes and associations of that class, as needed for computing routes and tracking vehicles.

The file contains the Vehicle class. Note that vehicle status is encoded with locationStatus as an enumerated type. In Umple, enumerated types and state machines share syntax. The difference is that the latter have events to change value (state) whereas a variable of an enumerated type has to have its values directly set.

Finally, the file contains the StationCoordinator class. In the base configuration (discussed in the next section) there would only be two: One FSC (Fire station coordinator) and one PSC (Police station coordinator). In other configurations there can be many.

```

/* BcmsCore.ump
 *
 * Core Umple model file for the Crisis
 * management system
 * Source document:
http://cserg0.site.uottawa.ca/cma2012/CaseStudy.pdf
 *
 * Author: Timothy C. Lethbridge
 *
 * To use this, a master file must be created
 * with use statements that link in
 * RoutesAndLocations.ump
 * and CoordinationStateMachine.ump
 * Plus provide multiplicities as described.
 */

/*
 * The singleton that will link to the various
 * system elements
 * Note that variants with different
 * multiplicities will associate this with
 * crises and coordinators
 */
class CrisisManagementSystem
{
    // There is a single crisis management system
    singleton;
}

/*
 * A crisis is what is being managed by the
 * Crisis Management System
 */
class Crisis
{
    identifier;
    Location gpsLocation;
    Time initialReportedTime;
    status {active {} closed {}}
    description;

    0..1 -> * Vehicle assignedFireVehicles;
    0..1 -> * Vehicle assignedPoliceVehicles;

    0..1 -> 0..1 Route proposedFireRoute;
    0..1 -> 0..1 Route proposedPoliceRoute;
    0..1 -> * Route disagreedRoutes;
    0..1 -> * RoadSegment;
        blockedOrCongestedSegments;
}

// Police or fire vehicle
// From Requirements section 8 Data Dictionary
class Vehicle
{
    vehicleID;
    Location currentLocation;

    Time eta;
    Location intendedLocation;
    locationStatus {
        station {}
        enRouteToCrisis {}
        atCrisis {}
        enrouteReturn {}
    }
}

```

```

0..1 -- 0..1 Route routePlan;
Boolean brokenDown;
Boolean inTrafficCongestion;
}

/*
 * The station coordinator is the automated
 * agent of the Human coordinator in
 * either the fire service (FSC) or police
 * service (PSC).
 * Any running version of this program will
 * just have one of these active as the
 * current coordinator and one or more active
 * as remote coordinators to communicate with
 */
class StationCoordinator
{
    identification;
    Boolean police; // false for fire
}

```

The above file can be obtained online as <http://try.umple.org?example=BcmsCore>

Figures 3 and 4 (discussed in the next section) show the UML class diagram for this, with two extra associations added,

2.4 Variant Models

One way of specifying variants in Umple is to have a series of top level ‘variant’ files, one per variant, that inject into various other classes the additional attributes, associations, states or other features needed for each variation point.

As an example the following are two variants. The first is the base configuration.

As implementation proceeds, variations of some of the methods that interact with the associations would also need to be specified in the same manner. Model-oriented programming makes this easy by handling methods in the same manner as other entities in a class.

Here is the first variant

```

/* BcmsBaseConfiguration.ump
 *
 * Master file to build the base version of
 * bcMS that has just one crisis and one
 * police coordinator and one fire coordinator.
 * This file requires the referenced
 * files to be present in the same directory.
 */
use RoutesAndLocations.ump;

namespace bcms;
use CoordinationStateMachine.ump;
use BcmsCore.ump;

class CrisisManagementSystem {
    1 -> 0..1 Crisis;
    1 -> 2 StationCoordinator;
}

```

Here is the second variant:

```

/* BcmsMultiCrisisMultiCoordConfiguration.ump
 *
 * Master file to build a version of

```

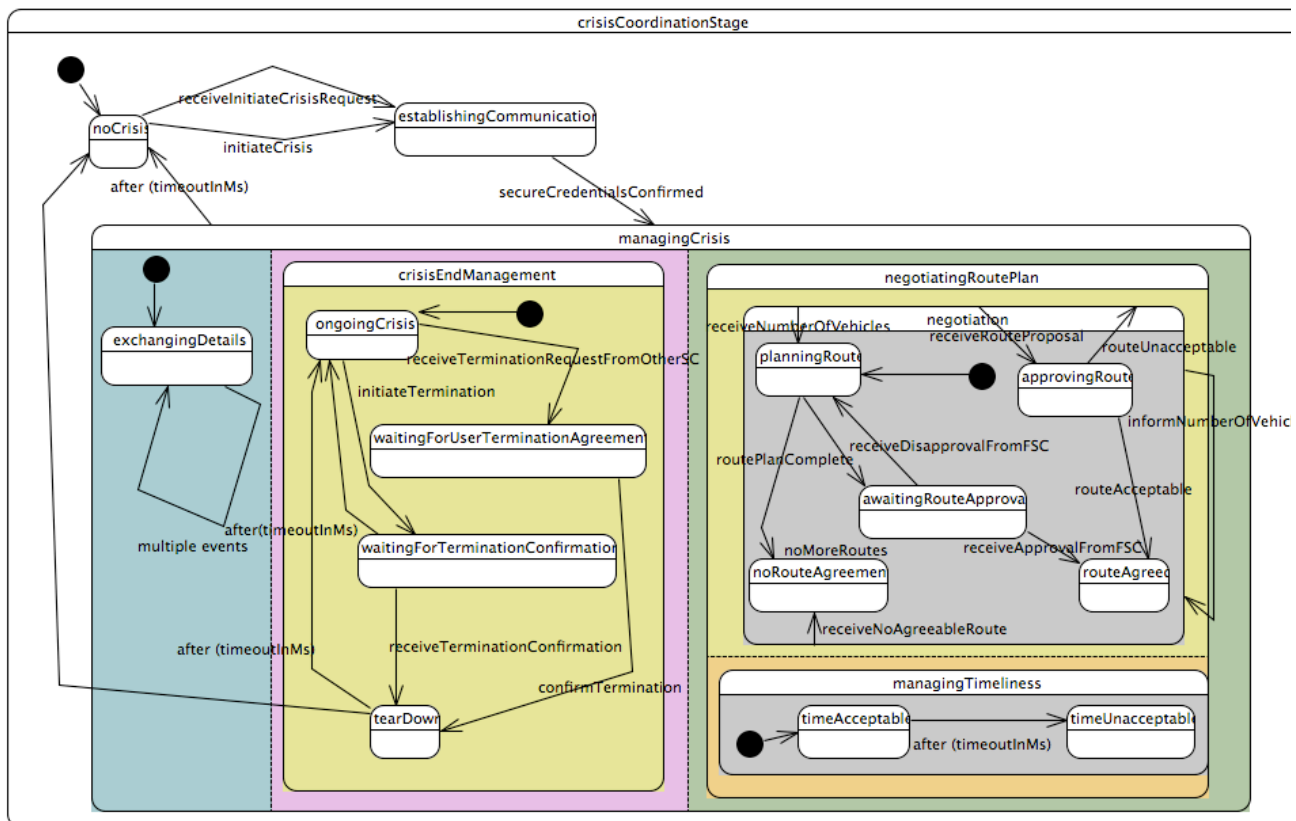


Fig. 2: State Diagram for Crisis States

```

* bCMS that has just many crises and
* coordinators
* This file requires the referenced
* files to be present in the same directory.
*/
use RoutesAndLocations.ump;
use CoordinationStateMachine.ump;
use BcmsCore.ump;

class CrisisManagementSystem {
  1 -> * Crisis;
  1 -> 2..* StationCoordinator;
}

class StationCoordinator {
  // Mixin to allow specification that one
  // is the lead
  Boolean lead;
}

```

Figure 3 shows the class diagram of the system core, with the first variant installed, generated by UmpleOnline. Figure 4 shows the second variant. The only difference is the multiplicities.

An alternative way to model variability with Umple is to use Umple's VML extension [10]. As with Umple, the VML tool can be used on the command line, but has an online version for demonstration purposes [11].

Below is the VML model. Change the last two lines to invoke different variants.

```

// bCMS.vml
// Variability model for various variants
// of the bCMS system
// Load at http://try.umple.org/vml.php

```

Concern Main

```

{
  VariationPoint core {
    Kind: Optional;
    use RoutesAndLocations.ump;
    use CoordinationStateMachine.ump;
    use BcmsCore.ump;
  }
}

```

Concern Multiplicity

```

{
  VariationPoint numCrises {
    Kind: Alternative;
    Variant single {
      class CrisisManagementSystem {
        1 -> 0..1 Crisis;
      }
    }

    Variant multiple {
      class CrisisManagementSystem {
        1 -> * Crisis;
      }
    }
  }
}

```

```

VariationPoint numStationCoordinators {
  Kind: Alternative;
  Variant pair {
    class CrisisManagementSystem {
      1 -> 2 StationCoordinator;
    }
  }
}

```

```

Variant multiple {
  class CrisisManagementSystem {

```

```

1 -> * StationCoordinator;
}
}

class StationCoordinator {
  Boolean lead;
}

}

// Invoke central system
invoke (Main,core);

// To invoke base case leave the following as
single and pair
// To change variants, change single and/or pair
to multiple
invoke (Multiplicity,numCrises,single);
invoke (Multiplicity,numStationCoordinators,pair);

```

Compiling using VML will produce the same result as just encoding the variants directly using the Umple files shown earlier.

The reader can view the VML file by going to VML online and selecting it from the Examples menu.

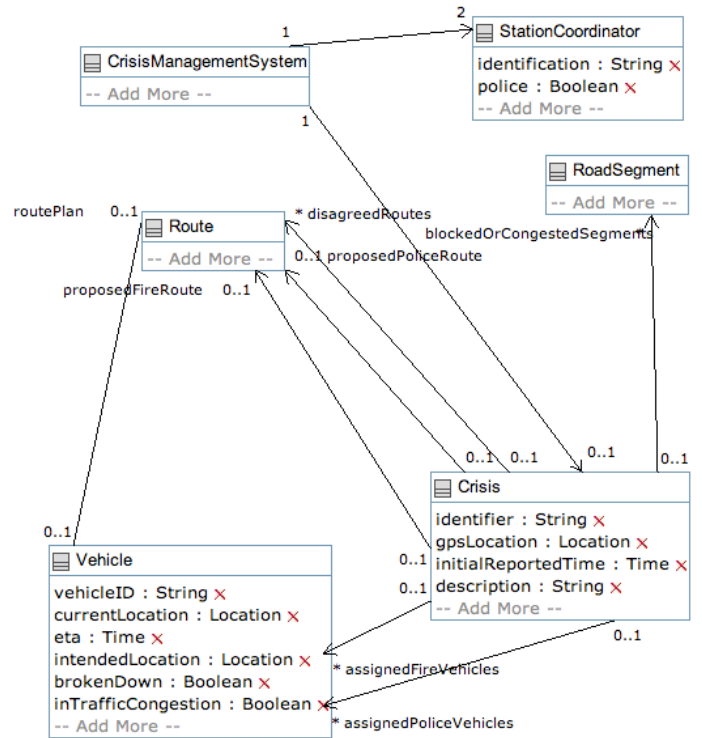


Fig. 3: System core with base variant

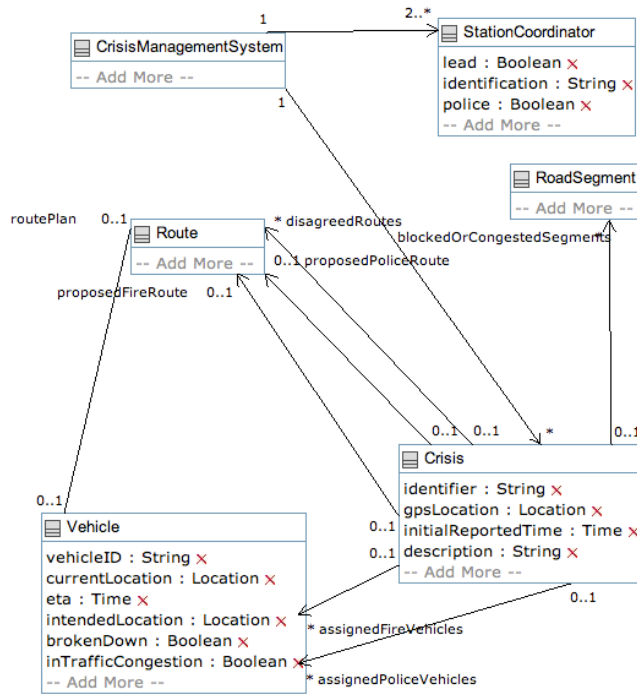


Fig. 4: System core with multi-crisis, multi-coordinator variant

3. REFERENCES

- [1] Afredo Capozucca, Betty H.C. Cheng, Geri Georg, Nicolas Guelfi, Paul Istioan, Gunter Mussbacher, "Requirements Definition Document For A Software Product Line Of Car Crash Management Systems", 2012, <http://cserg0.site.uottawa.ca/cma2012/CaseStudy.pdf>
- [2] Forward, A., Lethbridge, T.C., and Brestovansky, D. (2009), "Improving Program Comprehension by Enhancing Program Constructs: An Analysis of the Umple language", International Conference on Program Comprehension (ICPC) 2009, Vancouver, IEEE Computer Society, pp. 311-312.
- [3] CRuiSE Group, University of Ottawa, "Umple Model-Oriented Programming", <http://www.umple.org>, visited 2012
- [4] Forward, A., Badreddin, O., Lethbridge, T.C., Solano, J., (2011) "Model-Driven Rapid Prototyping with Umple", Software Practice and Experience, 42: pp. 781-707 DOI: 10.1002/spe.1155
- [5] Lethbridge, T.C., Forward, A. and Badreddin, O. (2010), "Umplification: Refactoring to Incrementally Add Abstraction to a Program", Working Conference on Reverse Engineering, Boston, October 2010, pp. 220-224
- [6] Neil Loughran, Pablo Sánchez, Alessandro Garcia and Lidia Fuentes, "Language Support for Managing Variability in Architectural Models", Springer LNCS, 2008, Volume 4954/2008, 36-51, DOI: 10.1007/978-3-540-78789-1_3
- [7] CRuiSE Group, University of Ottawa, "Umple Online", <http://try.umple.org>, visited 2012
- [8] Lethbridge, T., Mussbacher, G, Forward, A. and Badreddin, O. (2011) "Teaching UML Using Umple: Applying Model-Oriented Programming in the Classroom", CSEE&T 2011, pp. 421-428.
- [9] Lethbridge, T., and Laganière, R., *Object-Oriented Software Engineering: Practical Software Development Using UML and Java*, McGraw Hill, 2004
- [10] Levin, J, *System Generation for Time and Activity Management Product Lines*, Computer Science Masters Thesis, University of Ottawa, <http://www.site.uottawa.ca/~tcl/gradtheses/jlevin/>
- [11] Cruise Group, University of Ottawa, VML Online, <http://try.umple.org/vml.php>
- [12] Papyrus, <http://www.eclipse.org/modeling/mdt/papyrus/>
- [13] Badreddin, O., Forward, A., and Lethbridge, T. (2012), "Model Oriented Programming: An Empirical Study of Comprehension", Cascon, ACM, Accepted.